Jani Ahde

Real-time Unity Multiplayer Server Implementation

Bachelor's thesis
Information Technology / Game Programming

2017

Kaakkois-Suomen
ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tekijä	**Tutkinto**	**Aika**
Jani Ahde	Insinööri (AMK)	Marraskuu 2017
Opinnäytetyön nimi Reaaliaikaisen Unity-moninpelipalvelimen toteutus		58 sivua
Toimeksiantaja Nitro Games Plc (Oyj)		
Ohjaaja Lehtori Niina Mässeli		

Tiivistelmä

Tämä opinnäytetyö on pääasiallisesti toteutusdokumentti reaaliaikaisen ja omistautuneen pelipalvelimen tärkeimmistä komponenteista. Samalla käsittelyn kohteena on pelipalvelimen ottelulogiikka Nitro Games Plc:n (Oyj) kehittämässä ja hiljattain julkaistussa ilmaisessa mobiilipelissä nimeltä Medals of War.

Opinnäytetyössä käytettiin tutkivaa ja toteutuspainotettua lähestymistapaa. Projektin ensisijaisena työkaluna toimi pelien kehitysalusta Unity-pelimoottori, ja keskeisenä sovelluskehityksen menetelmänä ketterä ohjelmistonkehitys. Projektin edetessä useita verkkotyöskentelyyn ja Unityn sisäiseen verkkoteknologiaan perustuvia aiheita tutkittiin ja opiskeltiin laajamittaisesti. Pelipalvelimen runkona käytettiin Master Server Framework -kehikkoa nopeuttamaan palvelimen kehitysprosessia, ja haluttuja ominaisuuksia kehitettiin annettujen ohjelmistovaatimusten puitteissa.

Pelipalvelimen pääasiallisten ominaisuuksien kehittäminen oli projektissa pääpainona ja näin ollen palvelinputken prosessit toteutettiin ohjelmistovaatimusten ja pelin ominaisuuksien mukaisesti. Palvelimella pyörivien otteluiden logiikka toteutettiin Unityn oman verkkoteknologian avulla, ja toteutuksissa hyödynnettiin varsinkin pelimoottorin verkkoteknologian tarjoamia ominaisuuksia ja ohjelmointirajapintoja.

Opinnäytetyöprosessin aikana saavutettiin toimiva palvelinprototyyppi sekä julkaistuun peliin sovellettu pelipalvelin. Kehitetyn pelipalvelimen pohjatyö saatiin helposti jatkettavaksi ja jatkokehitettäväksi. Jatkokehitysprosessi käynnistettiin jo tämän opinnäytetyön loppuvaiheessa.

Asiasanat

Unity, Master Server Framework, barebones-masterserver, omistettu pelipalvelin, verkkotyöskentely, reaaliaikainen moninpeli, synkronisaatio

South-Eastern Finland University of Applied Sciences

Author	Degree	Time
Jani Ahde	Bachelor of Engineering	November 2017

Thesis Title

Real-time Unity Multiplayer Server Implementation

58 pages

Commissioned by

Nitro Games Plc

Supervisor

Niina Mässeli, Senior Lecturer

Abstract

This thesis is an implementation document of the key elements of a real-time dedicated game server and its match logic in regard to the Nitro Games Plc developed and published free-to-play mobile game Medals of War. Basic prototype and live game valid builds of the dedicated game server were developed at the time of this thesis.

Using a game development platform known as Unity Engine, many topics of Unity networking, networking in general and Master Server Framework were studied extensively, and therefore various findings related to these studies are conveyed through learned theories and practices. Master Server Framework is given particular attention due to being the essence of the dedicated game server. In regard to the vast network engineering field, conveyed theories are exclusively selected to support the implementation of the game server.

To support the development process, functional key requirements were provided to ensure implementation of vital server features such as validation and synchronization. Focusing largely on the core functionalities of the dedicated game server, various Master Server Framework specific topics are examined to provide concise understanding of the inner workings of the framework and the game server.

To provide match relational context, various match state affecting implementations of script components and network objects were included for more comprehensive view over the topic. Functional requirements were therefore emphasized on implementations when appropriate.

Extendable and functioning dedicated game server was produced as a result of the development process, marking the project as a success, and giving a basis for future implementations.

Keywords

Unity, Master Server Framework, barebones-masterserver, dedicated game server, networking, real-time multiplayer, synchronization

CONTENTS

ABBREVIATIONS 6
1 INTRODUCTION 7
1.1 Purpose of this thesis 7
1.2 Approach to the subject 8
1.3 Medals of War 8
1.4 Nitro Games Plc 9
2 FUNCTIONAL REQUIREMENTS 9
2.1 Network connection 10
2.2 Client input handling 10
2.3 Authority distribution 11
2.4 Action validation 11
3 THEORETICAL FRAMEWORK 12
3.1 Network 12
3.1.1 Network topology 12
3.1.2 Latency 13
3.2 Unity HLAPI 15
3.2.1 Remote actions 16
3.2.2 Web requests 17
3.3 Master Server Framework 18
3.3.1 MSF network 19
3.3.2 MSF peer 20
4 IMPLEMENTATION 21
4.1 Match concept 21
4.2 Game server core 22
4.2.1 Connection to the master server 24
4.2.2 Unet Game Room 27
4.2.3 Game server access 32

4.3 Networked objects 38
4.3.1 Player prefab 38
4.3.2 Match Manager object 39
4.3.3 Deployable cards 40
4.4 Game server match logic 40
4.4.1 Match Manager logic 41
4.4.2 Player object Squad 45
4.4.3 Characters and buildings 47
5 CONCLUSIONS 52
REFERENCES 54
LIST OF FIGURES

ABBREVIATIONS

CMD	Command
DOT	Data Transfer Object
HLAPI	High Level Application Programming Interface
HQ	Headquarters
LLAPI	Low Level Application Programming Interface
MSF	Master Server Framework
MVP	Minimum Viable Product
NDA	Non-Disclosure Agreement
NML	No Man’s Land
OSI	Open Systems Interconnection Reference Model
RPC	Remote Procedure Call
RTT	Round-Trip Time
SRS	Software Requirements Specifications
UNET	Unity Network Technology
UX	User Experience

1 INTRODUCTION

Modern mobile game development seems to heavily focus on social interactions, and multiplayer aspects of gaming. This can be realized if searched through either Apple App store or Google Play store. The focus on creating communities around games and having them create a social phenomenon can be seen on games like Supercell's Clash Royale, Plarium's Vikings: War of Clans and Wargaming's World of Tanks Blitz. Every one of these games implements multiplayer feature its own way, but a common factor is that they are using servers to allow this level of interaction.

1.1 Purpose of this thesis

The topic of this thesis is a Unity multiplayer game server and its development and implementation for a free-to-play mobile real-time strategy game called Medals of War which is currently, at the time of writing, being developed by a Finnish mobile game developer Nitro Games Plc.

This thesis is the means to study, document and describe the implementation of creating a multiplayer server for a mobile game using Unity, a game development platform, and its HLAPI system in together with Master Server Framework. The idea is to provide an easy to access document that conveys the theory behind the implementations and the explanations of those implementations used in the development of said server.

The main structure contains of the requirements of the server and what it should be capable and able to do. This is followed by introducing the theory behind networking and game servers in general. Unity HLAPI and the tools it provides for creating this server are also explained. Theory is followed by practice, where implementation is discussed and backed by theory and requirements previously handled. Implementation detail is the main chapter and focus of this thesis. After implementation is handled it is time to evaluate the development and implementation process and deduce what could have been done differently and what is still left to do.

1.2 Approach to the subject

The subject was approached with no prior knowledge of Unity's multiplayer implementations and thus all the epiphanies are a large combination of theory, prior knowledge of the existing employees of the company, and tremendous amount of research, trial, errors and experimenting.

Prior work related to multiplayer matches in the game were done with the mind on host-to-client multiplayer and minimum viable product. This gave a lot of ground work and structure for the server, but also introduced various parts that required reimplementation and redesign programming and functionality wise.

The work presented in this thesis lays open many key functionalities of the game server core and the match logic of the game Medals of War. When possible, pros and cons are considered with reasonings behind the implementation choices.

1.3 Medals of War

Medals of War is to be the newest addition to the mobile game portfolio of Nitro Games Plc and is a product of iteration heavy minimum viable product (MVP) process. During the writing of this thesis, the game had reached a soft-launch stage, and was available for download in Sweden and Netherlands from iTunes for iOS.

Medals of War itself, as previously mentioned, is a free-to-play real-time strategy game that is set into a World War 2 fantasy themed world of Warland. The game is community-focused, and almost every aspect of the game is designed to serve that purpose.

The revealed key features of the game are to collect various cards and Officers that are used to fill battle decks and to control the battlefield against the opponent in an intense player versus player battles and to compete against other player created communities (Medals of War 2017).

The most relevant aspect in the game, especially related to this thesis, is the player versus player feature that happens on Unity based game servers. The game servers are produced internally, and this thesis is a by-product of that process. Guidance of player connections to available game servers and deploying of said game servers is handled by the Master Server Framework, a Unity asset available from the Unity Asset Store.

The responsibilities of game servers are to handle the player input, produce the expected output based on said input, use player data to validate players' actions, and to deduce the match outcome, which allows correct distribution of player rewards.

1.4 Nitro Games Plc

Nitro Games Plc is a mobile game developer based in Finland, Kotka, and was founded 2007. The company's business entity type changed from a private company limited by shares to a public type during summer 2017, and was officially ringing the bell at Nasdaq First North Stockholm 16th of June for their first day of trading.

Nitro Games was during the writing of this thesis developing its new free-to-play mobile strategy game Medals of War. This thesis is heavily focused on the development and implementation of the Unity multiplayer server that is used to power every PvP match occurring in the game.

Nitro Games has previously focused on PC games but has since the release of Charlie Chucker, which was released on Windows Phone in 2013, focused on mobile games, and especially on free-to-play strategy games.

2 FUNCTIONAL REQUIREMENTS

Full-fledged document of the software requirements specifications (SRS) are out of the scope of this thesis but will not to be left out in entirety. It must be noted that the game server is not an independent software but more of a subset of an existing software. The following paragraphs should be adequate of giving a sufficient view of the key functional requirements of the Unity game

server that is used in the Medals of War.

2.1 Network connection

The game server is an entity with the purpose of being the meeting ground for two client instances of the game. From the match result validation point of view, the result of the match on the game server is the correct state and as such, the result for the clients. Implemented back-end side of the game will also be used to supplement the valid state of the ongoing matches.

When synchronizing the variable and state info of the match, the bandwidth available should not be cluttered or otherwise abused with constant state synchronizations. Instead, state synchronizations should be integrated into the events of the game that the game server perceives as important for the clients to replicate and synchronize their states into.

The game server should be able to handle received client connections with the ability to accept the clients into the game, store the clients' info for the duration of the match, and disconnect the clients as well. The client-side of the game used by the player should not crash or otherwise throw an exception when the connection is disconnected from the game server.

2.2 Client input handling

The state of match is required to be in precise knowledge of the game server, and the state that is present on the server must be the one used to determine the outcome of the match for all the connected clients. The invalid game state of the match present on the clients should be synchronized and forced by the server to match the server's state.

All client inputs that alter the state of the match, will be handled and validated by the game server. Resources used by the clients are resolved by the game server and all the actions the clients invoke without enough resources are deemed to be invalid and discarded.

All interactions between game entities are handled by the game server to preserve the integrity of the game state. Visual and audio feedback are to be left out from the game server for performance and no recipient reasons.

2.3 Authority distribution

Due to the competitive nature of the Medals of War, the clients will not able to gain any leverage or advantage against each other's through questionable actions performed through any third-party software or an external code.

The game server in Medals of War is an authoritative form of a server. Therefore, like previously mentioned, the state of the match and the variables that are significant to uphold this state, are most accurate and complete on the server. The game server dictates everything related to the match state, and enforces this by forcibly synchronizing its data onto the clients.

The players themselves will have client authority on their respective player objects. The clients will use their player objects to send commands, more easily perceived as inputs, to the game server, which in turn calculates the outcome and will either comply in accordance to the game logic or discard the player input as an invalid action.

2.4 Action validation

The game server will use the client sent data in the beginning of the match to cache the possible actions available to each client. This data is validated through the back-end server that has the knowledge of the available actions the player can do with his or her active battle deck.

The cards the clients use, are validated with the game server cached data. Invalid data is discarded and the player commands that were used to send the data are ignored. Invalid data will not initially require any other actions from the game server except ignoring and discarding the data. Receiving an invalid data will not penalize the client that sent the data as there is a chance that the data was corrupted or otherwise invalidated through a client-side error.

3 THEORETICAL FRAMEWORK

Due to the nature of this thesis and the complexity of the subject in hand, a certain amount of theory was required to achieve the quality Nitro Games required. Many of the terms and definitions used in this thesis are handled in this chapter.

The definition of a server can vary in many situations and especially in different context-scenarios. For simplicity and to avoid confusion, the game server can be called an instance of the game which simulates the matches and replicates the match state to the connected clients (Valve Developer Community 2001). In this thesis, the terms game server, server and dedicated server are used almost interchangeably. The term back-end server is used to describe the whole process that runs the database queries and handles the player corresponding data and the game configurations.

In this thesis, the game server is equivalent to the dedicated server. Dedicated servers are simply headless client instances void of rendering and user inputs and are hosted on a single machine or multiple server machines using multiple different programs to tie together (Weilbacher 2012).

3.1 Network

MVP version of the game was not using dedicated servers to run the course of the matches but trusted one of the clients to be a host. In a game where peer-to-peer or host-to-client network model is established, it is possible for the other peers or the host of the match to gain unfair advantage over the others through use of cheating software or, if the cheater is prone to use all tools available, through usage of a lag switch, especially if the host or the peer is favoring WLAN connection.

3.1.1 Network topology

Previously mentioned network topology is a way to define and describe the connection structure of each device and component in the network, and the topology can be viewed through logical or physical topology (Pandya 2013). Our perception is mostly logical.

Out of possible logical topologies, Unity HLAPI directly supports only two of them; those two being star and peer-to-peer topologies. The star topology can be divided into host-to-client and dedicated server-to-client networking models. In star topology, the network utilizes a central hub, a host or a game server in our case, and all the other devices are connected to that server or host. In server-to-client network, the failure of one client is not disruptive to the functionality of the server and other clients. This gives reliability on the multiplayer matches. However, in the case that the game server fails, the match is unresolved, and the clients must failsafe their way out the situation (Pandya 2013).

In peer-to-peer topology the main idea is to give each client control over the running game state, and to allow each peer's device to simulate the game state through dividing the network and resources in an evenly manner and synchronized pattern (Schollmeier 2002).

To best serve the features of the game server, the logical network topology chosen was star topology, and more precisely server-to-client model. This serves and fulfills the concept of the dedicated game server. Consequently, this allow the largest possible integrity and amount of control over the game state. Furthermore, authoritative model of the server mitigates with possible vulnerabilities of the match state.

3.1.2 Latency

Released games have to contend with various negative factors that may or may not be present during the development. When networked games are not played in the development local network, a negative factor known as latency must be taken into account. Technically the word latency is referred to as perceivable time and delay between the client issued action and the outcome shown on the client after being processed through the game server (Glazer & Madhav 2015).

Latency is measured through the round-trip time (RTT), which consists of the time of a sent packet from a peer to another and the time it takes for that peer to receive a response packet. However, RTT is not only a combination of

forming, sending and receiving times of network packages, but frame rate and computing time to handle and deserialize the package, are factors as well (Glazer & Madhav 2015).

Figure 1 illustrates package sending and receiving on both, the server and the clients. To depict RTT, the client can send a CMD package to the server, conveying the client's input and the desired action.

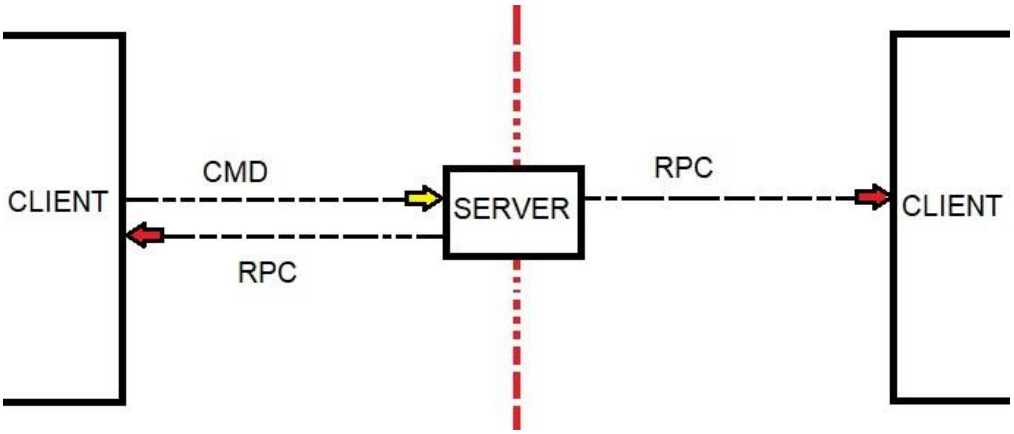

Figure 1. CMD package.

As the latency and the RTT are not constant values, the next RTT measurement example uses arbitrary but reasonable values. If deducing the client's input and the wanted action takes 15 ms and the amount of time it would take for the action consisting CMD package to reach was 30 ms, then the first half of the round would be 45 ms. The next round would consist of server's deduction and computation of the wanted action, its validity and possibility related to the current state of the server state. Reasonable time assumption, depending on the action, could be 15 ms, and fulfilling the command another 15 ms, summing up to a total of 30 ms on the server-side. Having updated the server state, the refreshed state with the response related to the received action consisting package, could take the same amount of transfer time as it did from the client to the server, that is 30 ms. To sum up the whole round, the first round being 45 ms, and the second being 60 ms, the whole RTT would be 105 ms. Thus, RTT consists of the computation and over the network transfer times.

Low RTT and latency are not self-evident however. In a situation where it can be expected or there is a possibility that the RTT stays low, there rises a requirement to stem the problem through reducing the discrepancy of the response times through lag compensation, which effectively means either the

client-side or the server-side lag compensation implementations (Xicota 2013).

It must also be taken into consideration that the latency is not just hardware and software related. When measuring the elapsed time for something to happen through the latency, the latency can be stated to be some arbitrary value. However, the perceivable delay in the latency is just as psychological as it is a measurable number, and different actions have a certain degree of acceptable latencies in contrast to others (Ignatchenko 2017).

3.2 Unity HLAPI

The main framework used for building the game server of this thesis, is Unity's engineered and implemented HLAPI. HLAPI was designed and built on top of the OSI-models transport real-time communication layer, which would allow it to handle various common tasks related to the multiplayer capabilities of multiplayer games (Unity's HLAPI 2017). Match logic of the game server is solely based on the HLAPI provided functionalities.

All networked and synchronized match objects are spawned through the game server. When an object is spawned, HLAPI will assign an available NetworkIdentity and a valid authority model to the object. Spawned objects with their serializable values are instantiated to all the available clients and the server (Unity's Object Spawning 2017). The NetworkIdentity assigned to the spawned object is unique and identical through the network participants and can be used as a tool for action validations.

Spawnable objects are required to be registered to the active NetworkManager. Spawnable objects are handled by the spawn manager, NetworkServer and ClientScene classes during the spawning process. NetworkServer runs an internal update method which compares the dirty bit of each NetworkIdentity containing object. Objects that are marked dirty, build UpdateVars packages that are forwarded by the server to each client's corresponding object and update SyncVar attributed variables (Unity's State Synchronization 2017).

SyncVar attributed variables can be attached with hooks. Synchronization

hooks take the new value of the variable as an argument and are only called on the clients. To illustrate, if the client instance is bound to launch an event, such as a sound at a certain value of the hook attached variable, the hook method can launch the event and refresh the variable in the same call. Using the hooks to launch client-side only methods is a convenience.

Due to an authoritative model of the game server, the SyncVar attributed variables offer an additional security layer. Automatically forwarded UpdateVars packages are dominant with their variable states and will override the client-side values by the default HLAPI implementation, thus rendering the tampering of the said values futile.

However, the SyncVar variables cause an additional load for the game and the network in general. Unity HLAPI presents a hard cap of 32 SyncVar attributed variables on a class before starting to generate UNET compilation errors. The limit can be raised, but before proceeding, it might be best to rethink the design of the class requiring that many SyncVar variables.

3.2.1 Remote actions

SyncVar packages are limited in their capabilities of transferring data due to their restrictions to be able to be declared on only the basic built-in types such as byte, Vector3 and structs containing the allowed types. To preserve the match integrity, various match events require more comprehensive methods for synchronizing these events.

Unity HLAPI is provided with three types of remote actions. When the client wants to affect the match state, the game server can’t comply without proper validation. The clients can affect the game state through actions called commands in the HLAPI and they can only be sent from the client authority containing player objects and, after Unity 5.2, client authority containing non-player-objects. However, UNET disallows player object deprived client connections from sending commands, allowing only the basic network messages being transmitted. There are restrictions to the command methods as well. Methods that are commands can’t be static methods and must be prefixed with a keyword *Cmd* and attributed with the *[Command]* attribute. Command

methods can have parameters of basic types and HLAPI relative networking variables or structs consisting of the listed basic types (Unity's CommandAttribute 2017).

Unity serializes the given arguments into byte arrays that are transmitted with the command to the corresponding active game server instance. This allows the command to be invoked on the server with the exact same values as it was called on the invoking client. UNET doesn't differentiate if the command is initially invoked on the client or on the server due to commands are only executed on the server-side.

The other two remote actions provided grant network server instances the capability to invoke ClientRpc and TargetRpc attributed methods that will be called on the server but invoked on both clients (ClientRpc) or a specified client (TargetRpc). RPC methods must be prefixed with a *Rpc* and attributed with *[ClientRpc]* or *[TargetRpc]*. Like commands, client replication methods can have parameters of basic types, HLAPI relative network variables and structs of listed basic types (Unity's ClientRpcAttribute 2017).

3.2.2 Web requests

All the client data can't be trusted to be provided by the clients and as such, requires a neutral, trustworthy participant, such as a database. The database stores all the player data in a neutral form and can be accessed through the back-end. As the back-end is not locally available for the game server, the connection must be made through a HTTP request.

HLAPI allows the forming and handling of HTTP or HTTPS requests and responses through a UnityWebRequest, which is working through the HLAPI and LLAPI. The LLAPI allows the developers to implement customized UnityWebRequests, UploadHandlers and DownloadHandlers for specific scenarios that the default requests can't handle as required (Unity's Advanced operations: Using the LLAPI 2017).

Basic retrieval of data from the database and through the back-end, can be

achieved through a UnityWebRequest's GET method which allows Unity instances to connect to any web service through the previously mentioned protocols. In the contrary, a UnityWebRequest's POST method allows the sending of data to the back-end for storing purposes. Responses from both requests can be handled through attachable callback methods.

The response from the request can be handled through Unity's coroutines by yielding UnityWebRequests Send (Unity 2017.1 and downwards) or SendWebRequest (Unity 2017.2 and upwards) methods as they will set the coroutine on a pause until system error or finished communications (Unity's SendWebRequest 2017), or by creating a fully customized class inheriting a Unity provided DownloadHandlerScript, which has various callbacks for when the respond has arrived for the request (Unity's Creating DownloadHandlers 2017).

3.3 Master Server Framework

Master Server Framework is an API and a framework designed by Alvydas Poškevičius that is barebones for creating the logic and implementation of the server pipeline for Unity multiplayer games and its fully open-source and freely downloadable as a Unity asset from the Unity asset store or the GitHub repository (Poškevičius 2017).

MSF provides a server pipeline framework and a couple of demos for networked games with three differently tasked server types for maximum scalability and distribution of logic and implementation: master server, spawner server and game server.

The framework has an emphasis on a root level master server where all the instances of the game and the server infrastructure connect as can be seen in Figure 2. The master server can direct clients to available game servers or assign a spawner server to create a new game server instance for the clients to connect into (Poškevičius 2017). This form of functionality can also be available to various forms of matchmaking servers, as they can direct the clients to available game instances with different parameters such as any game available or a game that matches with the possibly provided comparable variables (Ignatchenko 2017, 32).

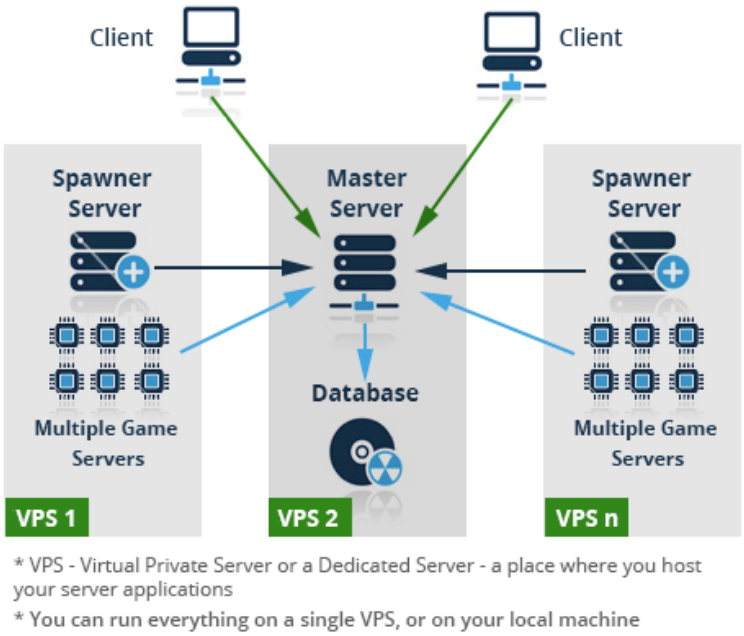

Figure 2. The server structure and pipeline of MSF. The Big Picture (Poškevičius 2017).

Spawner servers are mainly connected to the master server, and are a tool to extend and scale the availability of the game server instances when required. In a scenario where a client is querying for a game server and there is none to offer, a spawner server is assigned by the master server to boot a game server instance and inform the master server when the game server's game room is available for the players (Poškevičius 2017).

As the MSF is only a framework, it doesn't offer a full server infrastructure and pipeline solution for games, and requires extensive knowledge of C# and UNET programming (Poškevičius 2017). MSF offers a lot of boilerplate code and default modules but encourages developers to create their own implementations. It must be noted that the MSF is not only about programming, it requires server deployment knowledge to be properly deployed for usage in a commercial game.

3.3.1 MSF network

MSF provides its own network API to differentiate itself from the UNET implementation and to offer customizable and different solutions. However, it does not discard Unity as it is meant to be developed within Unity. Various layers of abstractions are meant to be efficient and conveniently extendable. Two connection protocols are provided with the MSF, called Websockets/TCP, which

is based on websocket-sharp, and UDP/RUDP, which is based on UNET Network Transport Layer API (Barebones-Masterserver's Networking API 2017). Both protocols are socket based and are meant to establish a connection between two endpoints. In the MSF, one of the endpoints must be of an interface client type IClientSocket, and other of a server type IServerSocket. The connection participants will perceive each other as peers. In the MSF peers are perceived as an interface type IPeers and every connection has the peer information embedded into them (Barebones-Masterserver's Networking API 2017).

Each peer can send network messages, identified as of an interface type IMessage, and receive network messages, identified as of an interface type IIncomingMessage. Each network message is related to the connection between the endpoints, and expose multiple properties and events to tap into. All MSF network messages are identified through specified message handlers that are set to work with different MsfOpCode containing network messages. MsfOpCodes are enumerations of aptly identified types of network messages such as the RegisterRoom and the GetRoomAccess.

3.3.2 MSF peer

Technically peers are important entities in the MSF. Peers are conceptually entities with corresponding layers such as different machines, software, human beings and hardware (Tanebaum & Wetherhall 2010). Connections and messages between differently characterized game instances happen between peers. In the MSF, each peer is represented through the IPeer interface.

IPeer is an interface that is used by the MSF server objects to identify and control the data available to or from an entity that in this case can be called as a client, which in the context of the MSF can be a player connecting to a game server, a spawner server or anything that is connected to the master server or even the master server itself. (Barebones-Masterserver's Peer Extensions and Properties 2017). In the MSF, between client-server connection, the client perceives the server as a peer while the server perceives the client as a peer.

Using the IPeer interface allows the server to cache all the unique peers currently connected to it. Each active peer on the server has a state and can have various additional privileges or different methods of handling through interface extensions. In the game server's case, when the master server receives a connection from the game server in an attempt to register itself into the master server, the IPeer of the game server can for example be provided with the game room extensions that contain customized handlers and properties for the game room identified IPeer. In addition to having differentiated interface extensions on the master server, the game server can do the same for its peers, the client players. The IPeers connected to the server have a state that the server can modify. IPeer state is both, indicated through IPeer properties and interface extensions.

4 IMPLEMENTATION

The development and implementation of the game server was started during the end of April and has not ceased during the writing of this thesis. Various components of the game server are described in this chapter, and all the main components and sub components are split to their own perspective subsections. To understand what the explained implementations are about, one must understand the match concept.

4.1 Match concept

Ranked games, or networked matches as interchangeably referred in this thesis, are the cause and the reasoning behind the need for the game server on the Medals of War. The games are meant to be competitive and fast paced with strategic and tactical elements fit to the genre of free-to-play real-time multiplayer strategy games.

Matches are held between two client players each holding their own base, referred to as Headquarters (HQ), on the corners of opposing sides of the map. These HQs are the main lifeforce and most conclusive objects when calculating the winning side of the match. Although being the game world representative of the players presence in the match, each HQ has the capability of launching a devastating missile fire against opposing HQ if the player has the

control of two capturable areas, located at the center of the map, called No Man's Land's sections.

The No Man's Land (NML) is one of the driving forces during the match as the control of the area guarantees constant damage against the opposing player's HQ. In some situations, the players can be equally matched, and no conclusive outcome seem reasonable in that situation. The games are timed and during the writing of this thesis, the matches game time limit was set to three minutes through the database. If the time runs out the match ends. If the time runs out and both HQs are alive and both NML's sections are without owners, the match results into a draw and no rewards are distributed. In time out situation, if one of the players own both the NML's sections, that player wins.

Available actions are measured through manpower and command points. Actions can be perceived as calling infantry, vehicles and commands into the battlefield. Manpower generates steadily as the match progresses, though during the last minute of the game, the generating speed is doubled. Command points are generated through drawing of non-command cards. Commands can be anything from calling a squad of paratroopers, calling down naval bombardments or plain establishing of a foothold through placing buildings in the battlefield. Each card has its own cost and work better against certain types of cards than others. Manpower is capped at the score of ten, while thee command points are capped to the number five.

Drawing cards is the only way to progress during the match as most interactions between the game entities are done through the game world representatives of the cards used by the players. Thus, it could be said that the main core loop of the game is to draw cards, try to control the NML's sections and to finally destroy the opponent's HQ.

4.2 Game server core

To provide a starting point for the required functionalities of the game server, a prefab containing the GameServer script component was created and attached to the required scenes as a not destroyable object. To have one access point for handling match and master server connections offered simplicity

and a starting point for the game server.

The initialization, as seen in the Figure 3, and the release of the attached connections were implemented into the MonoBehaviour derived Awake, Start and OnDestroy methods. Each GameServer instance, is a singleton object. In the Singleton pattern, a class has a single instance active and a single access point from the global namespace of the code during the lifetime of the program (Nystrom, 2014).

```
1.          void Awake()
2.          {
3.              if (!CreateSingleton())
4.                  return;
5.
6.  #if SERVER
7.              IsServer = true;
8.  #endif
9.              if(ForceServer || IsHeadless())
10.                 IsServer = true;
11.
12.             UnityEngine.Debug.Log("GameServer.Awake() IsServer: " + (IsServer ? "TRUE" : "FALSE"));
13.
14.             if(m_ConnectionToMaster != null)
15.                 m_ConnectionToMaster.ConnectOnStart = false;//IsServer;
16.
17.             Msf.Connection.Connected += OnConnected;
18.             Msf.Connection.Disconnected += OnDisconnected;
19.
20.             if(IsServer)
21.                 Destroy(m_ClientObject);
22.             else
23.                 Destroy(m_ServerObject);
24.
25.             m_GameRoom = m_GameRoom ?? FindObjectOfType<UnetGameRoom>();
26.
27.             m_MatchMaker = m_MatchMaker ?? FindObjectOfType<MatchMaker>();
28.         }
```

Figure 3. GameServer Awake method.

Awake method was implemented to find the UnetGameRoom and the MatchMaker component references and to determine whether the instance of the game is a headless game server or not. This is important to branch the logic between the client and the dedicated server. The connections to the MSF delegates are populated during this sequence, and depending on the server status of the game instance, either the client or server objects are discarded from the memory.

4.2.1 Connection to the master server

The GameServer component itself is not connected to the match logic in any way. The task of the GameServer is to connect to the correct master server when booted up and it is to handle the connections of the active clients during the match. GameServer connection to the master server is initiated by fetching the configuration data from the command line or the back-end as seen in Figure 4.

```
1.  IEnumerator WaitConfigs()
2.  {
3.      while (!ConfigData.ConfigsLoaded())
4.      {
5.          yield return null;
6.      }
7.
8.      // Default master server
9.      if (!Msf.Args.IsProvided(Msf.Args.Names.MasterIp))
10.         m_ConnectionToMaster.ServerIp = ServerConfig.GetConfigValue("IP_VALUE_KEY", m_ConnectionToMaster.ServerIp);
11.     if (!Msf.Args.IsProvided(Msf.Args.Names.MasterPort))
12.         m_ConnectionToMaster.ServerPort = ServerConfig.GetConfigValue("PORT_VALUE_KEY", m_ConnectionToMaster.ServerPort);
13.
14.     // Game Server to connects to master right away
15.     if(IsServer)
16.         ConnectToMaster();
17.
18.     if(!IsServer)
19.         StartCoroutine(ChooseMaster());
20. }
```

Figure 4. GameServer WaitConfigs coroutine.

To handle the connection to the master server, the server's port and the IP are required. Due to various situations, these two can change, which is why the correct values are best left to be fetched from the back-end. It is possible that all the game configs are not loaded when trying to connect to the master server, which is why coroutines are utilized to guarantee all the game data is available before choosing the master server and invoking connection.
The back-end provides a country specified IP address and a port number for the clients when trying to connect to the master server. The game servers have these parameters provided through a spawner server or direct command line arguments.

Connecting to the master server is an asynchronous task due to not being an instantaneous task. The master server must accept the connection before it is established. The initialization of the connection works the same way for both, the client and the game server and can be seen in Figure 5.

```
1.  private IEnumerator InitConnection()
2.  {
3.      // Wait a fraction of a second, in case we're also starting a master server.
4.      yield return new WaitForSeconds(0.2f);
5.
6.      var connection = GetConnection();
7.      TimeToConnect = MinTimeToConnect;
8.
9.      while (true)
10.     {
11.         if (connection.IsConnected)
12.             yield break;
13.
14.         connection.Connect(ServerIp, ServerPort, Mathf.CeilToInt(TimeToConnect * 1000.0f));
15.
16.         bool connected = false;
17.         bool failure = false;
18.         connection.WaitConnection((socket) =>
19.         {
20.             if (socket == null || socket.IsConnected == false)
21.             {
22.                 failure = true;
23.                 if (socket != null)
24.                     socket.Disconnect();
25.             }
26.             else
27.                 connected = true;
28.         }, TimeToConnect);
29.
30.         // Wait until one of the flags is set.
31.         while (connected == false && failure == false)
32.             yield return null;
33.
34.         if (connected)
35.             yield break;
36.
37.         TimeToConnect = Mathf.Min(TimeToConnect * 2, MaxTimeToConnect);
38.     }
39. }
```

Figure 5. ConnectionToMaster's InitConnection coroutine.

A lot depended on the geographical location of the master server and the available connection of the mobile device in use when initiated from the client. Game servers do not have this issue due to being spawned geographically to the exact same location as the master server. This geographical factor did cause some unfortunate issues on the clients though. Initially the coroutine was built to initiate the connection and wait for the response from the master server until the connection timeout threshold. In many situations, multiple connections to the master server could be established due to varying response times, which initially led to disconnections. It was best to create connected and failure flags to allow better distinction of the actual state of the connection. Connection result was set to be caught through an anonymous lambda

method. If met by a failure, the clients didn't stop the connection routine automatically and this was corrected. To remedy multiple connections, each connection attempt was given a certain time before trying again.
The game server can be manually booted standby instance waiting for the clients to join, or it can be a spawned instance through the master server called spawner server. Both possibilities need to be handled accordingly when the game server has connected to the master server as can be seen in Figure 6.

```
1.      private void OnConnected()
2.      {
3.          if (Msf.Server.Spawners.IsSpawnedProccess)
4.              StartSpawned ();
5.          else
6.          {
7.              if(Msf.Args.IsProvided(Msf.Args.Names.AssignedPort))
8.                  NetworkManager.Instance.networkPort = Msf.Args.AssignedPort;
9.
10.             NetworkManager.Instance.StartServer ();
11.         }
12.     }
```

Figure 6. GameServer's OnConnected method.

Having started the game server as a standby instance, which is waiting for clients to join, can be done without further settings. If the listener port for the game server has been supplied through the command line arguments, then that port will be used and the server listening routines are started as implemented in the Unity HLAPI. Standby instances of the game server can also be used during the development as sort of a debug game servers as they can be debugged locally. It is definitely faster to boot up developers own game server up and connect to that one with the client instances, instead of hoping to connect through the normal initiation of ranked matches on the clients where matchmaking and room searches are utilized.

Many game servers are simply spawned processes and are started on an as-needed basis. These instances are required to connect to the master server and register as a spawned process. Spawn task controller is constructed and cached from the response into the game server instance's local UnetGameRoom as shown in Figure 7.

```
protected virtual void StartSpawned()
{
    Msf.Server.Spawners.RegisterSpawnedProcess(Msf.Args.SpawnId, Msf.Args.SpawnCode, (controller, error) =>
    {
        if (controller == null)
            throw new System.Exception("Failed to register a spawned process: " + error);

        // Set the static object, so that it can be used when creating a room.
        UnetGameRoom.SpawnTaskController = controller;

        // Use the assigned port from command line arguments.
        NetworkManager.Instance.networkPort = Msf.Args.AssignedPort;
        NetworkManager.Instance.StartServer();
    });
}
```

Figure 7. ConnectionToMaster's StartSpawned method.

UnetGameRoom handles the game room registering, access into the game server and caches the match properties provided through the spawn task controller further down the pipeline. Spawn task controllers are used to relay data, instantiate and kill game server processes. Default settings provided with the MSF are adequate for the job.

4.2.2 Unet Game Room

Unity HLAPI provides an internal system for building staging areas for multiplayer matches through a networking class called NetworkLobbyManager which is derived from the NetworkManager class (Unity's NetworkLobbyManager 2017). Lobbies could be described as game rooms prior to the actual multiplayer match where the rooms have a limited number of open positions available for the match joining players. MSF does, however, make a difference between the functionalities of a lobby and a game room. In terms of having a competent multiplayer match, the lobby or the game room waits that it has enough players and all the players are ready before initiating the match. MSF provides the UnetGameRoom component that is designed to work with the other MSF provided components and functionalities.

Important for the UnetGameRoom component is to register itself as an available game room for the master server, handle client access to the game server and invoke methods attached to the system actions related to player joining and player leaving. Initiating the registration can be seen in Figure 8.

```
1.  // This will be called, when NetworkManager.StartServer is called.
2.  public override void OnStartServer()
3.  {
4.      // Find the manager, in case it was inaccessible on awake.
5.      NetworkManager = NetworkManager ?? FindObjectOfType<NetworkManager>();
6.
7.      // The Unet server is started, we need to register a Room
8.      BeforeRegisteringRoom();
9.      RegisterRoom();
10. }
```

Figure 8. UnetGameRoom's OnStartServer method.

Connecting to the master server allows the game server to register itself as a valid game server ready to receive connection from the clients and start a match. Unity allows the overriding of the OnStartServer method that is called when the HLAPI starts the server process of the game instance when called as can be seen in Figure 8. Pre-registration and registration of the game room calls are called through this method as well, and depicted in Figure 9.

```
1.  protected virtual void BeforeRegisteringRoom()
2.  {
3.      if (SpawnTaskController != null)
4.      {
5.          // If this server was spawned, try to read some of the properties
6.          var prop = SpawnTaskController.Properties;
7.
8.          // Try to find required properties through the key comparisons.
9.          // Keys can vary from number of players to scene names etc.
10.         if (prop.ContainsKey(MsfDictKeys.PROPERTY_0_KEY))
11.             PROPERTY_0 = prop[MsfDictKeys.PROPERTY_0_KEY];
12.     }
13.     #if UNITY_EDITOR
14.     else
15.     {
16.         // Debug or fail-safe parameters and properties can be hard-coded here.
17.         // Debug booted game server is not spawned and will always run this clause.
18.     }
19.     #endif
20.
21.     // Override the default public address.
22.     if (Msf.Args.IsProvided(Msf.Args.IP_PARAMETER) && NetworkManager != null)
23.         PublicIp = Msf.Args.IP_PARAMETER;
24. }
```

Figure 9. UnetGameRoom's BeforeRegisteringRoom method.

Before registration of the game room, the spawner of the game server might have provided properties for the game server to cache and to abide by. If the game server was started as a debug game server, the Unity Editor preprocessor region can be used for setting up the game server with the wanted properties. Possible properties are provided through the specified argument and

property keys with their own respective values as defined in the SpawnTaskController. MSF comes with a few predefined keys, and adding or deleting keys was fundamentally modifying the declaration of the MsfDictKeys class.

```
1.      public virtual void RegisterRoom()
2.      {
3.          var properties = (SpawnTaskController != null)
4.              ? SpawnTaskController.Properties
5.              : new Dictionary<string, string>();
6.
7.          if (properties.ContainsKey(MsfDictKeys.MAP_KEY) == false)
8.              properties[MsfDictKeys.MAP_KEY] = FAILSAFE_MAP;
9.
10.         // 1. Create options object
11.         var options = new RoomOptions()
12.         {
13.             // Set variables such as RoomIp, RoomPort, RoomName here.
14.             // Set IsPublic as true due not using MSF lobbies.
15.             IsPublic = true;
16.             AllowUsersRequestAccess = DEFAULT_VALUE,
17.             Properties = new Dictionary<string, string>()
18.             { {MsfDictKeys.MAP_KEY, MAP_VARIABLE} // Store the map name. }
19.         };
20.
21.         #if UNITY_EDITOR
22.         // Set debug values as UNITY_EDITOR is set only when running debug game servers.
23.         options.Properties.Add(MsfDictKeys.RANK_REQUIREMENT_KEY, "0");
24.         options.Properties.Add(MsfDictKeys.SCORE_REQUIREMENT_KEY, "0");
25.         #endif
26.
27.         BeforeSendingRegistrationOptions(options);
28.         // 2. Send a request to create a room
29.         Msf.Server.Rooms.RegisterRoom(options, (controller, error) =>
30.         {
31.             if (controller == null)
32.                 return;
33.
34.             Controller = controller;
35.             OnRoomRegistered(controller);
36.         });
37.     }
```

Figure 10. UnetGameRoom's RegisterRoom method.

Concluding the pre-registration phase initiates the actual registration sequence (Figure 10) where all the properties that define the match, which is run by the game server, are cached into the network serializable class called RoomOptions. Cached properties in the RoomOptions is sent through to the MsfRoomsServer as can be seen in Figure 11.

```
1.      public void RegisterRoom(RoomOptions, RoomCreationCallback)
2.      {
3.          if (Connection.IsConnected == false)
4.          {
5.              callback.Invoke(null, "Not connected");
6.              return;
7.          }
8.
9.          Connection.SendMessage((short) MsfOpCodes.RegisterRoom, roomOptions, (status, response) =>
10.         {
11.             if (status != ResponseStatus.Success)
12.             {
13.                 // Failed to register room
14.                 callback.Invoke(null, response.AsString("Unknown Error"));
15.                 return;
16.             }
17.
18.             int roomId = response.AsInt();
19.             RoomController controller = new RoomController(roomId, Connection, roomOptions);
20.
21.             // Save the reference of the RoomController.
22.             _localCreatedRooms[roomId] = controller;
23.
24.             callback.Invoke(controller, null);
25.             if (RoomRegistered != null)
26.                 RoomRegistered.Invoke(controller);
27.         });
28.     }
```

Figure 11. MsfRoomsServer's RegisterRoom method.

To register the room, the MsfOpCodes.RegisterRoom message is sent with the RoomOptions constructed as seen in Figure 10. RoomOptions class is a serializable data transfer object (DTO) class with overridden to-binary and from-binary writers that are used for serialization and deserialization. RoomOptions hold the properties of the game server's game room. RoomController, which is built with the essential master server identifiable room id, is used to handle the access to the game server and to notify the master server of changes to the game server properties and the player connections. With both, the RoomOptions and the RoomController, the default implementations of the classes are sufficient and don't require significant changes.

To handle the MsfOpCodes.RegisterRoom message, the master server is required to utilize message handler for that specific MsfOpCode. Handling of the RegisterRoom message is illustrated in Figure 12.

```
1.      private void HandleRegisterRoom(IIncommingMessage message)
2.      {
3.          if (HasRoomRegistrationPermissions(message.Peer) == false)
4.          {
5.              message.Respond("Insufficient Permission Level!", ResponseStatus.Unauthorized);
6.              return;
7.          }
8.
9.          RoomOptions options = message.Deserialize(new RoomOptions());
10.
11.         RegisteredRoom room = RegisterRoom(message.Peer, options);
12.
13.         // Callbacked roomId is used to create the master server accessible RoomController.
14.         message.Respond(room.RoomId, ResponseStatus.Success);
15.     }
```

Figure 12. RoomsModule's HandleRegisterRoom method.

As the final step of registering the game server's game room available for the master server to distribute, is to simply check that the game server instance is valid with its registration permissions. Having passed the checkpoint, the RoomOptions are cached and constructed as a RegisteredRoom which represents the game servers game rooms on the master server. To construct the RegisteredRoom from the cached RoomOptions, a public virtual method named RegisterRoom is utilized and implemented as in Figure 13.

```
1.      public virtual RegisteredRoom RegisterRoom(IPeer peer, RoomOptions options)
2.      {
3.          RegisteredRoom room = new RegisteredRoom(GenerateRoomId(), peer, options);
4.
5.          var peerRooms = peer.GetProperty((int) MsfPropCodes.RegisteredRooms) as Dictionary<int, RegisteredRoom>;
6.          if (peerRooms == null)
7.          {
8.              // Save the dictionary.
9.              peerRooms = new Dictionary<int, RegisteredRoom>();
10.             peer.SetProperty((int)MsfPropCodes.RegisteredRooms, peerRooms);
11.             peer.Disconnected += OnRegisteredPeerDisconnect;
12.         }
13.
14.         // Add a new room to peer.
15.         peerRooms[room.RoomId] = room;
16.         // Add the room to a list of all rooms.
17.         Rooms[room.RoomId] = room;
18.
19.         if (RoomRegistered != null)
20.             RoomRegistered.Invoke(room);
21.
22.         return room;
23.     }
```

Figure 13. RoomModule's RegisterRoom method.

RegisterRoom method as seen above, takes an IPeer, which in this case is the game server instance, and RoomOptions arguments. RegisteredRoom handles the access requests to the room by communicating with the game server's RoomController and represents the game server on the master

server. MSF does allow multiple game rooms on a single game server instance if necessary. The performance could, however, take a hit from such endeavor and ensuring the functionality could have been arduous. Therefore, such was not implemented. Each game server instance has only one game room in the current implementation and as such, the IPeer passed into the method doesn't have already registered rooms in its properties. The fresh master server registered room is also registered for the game server's peer data as well as into the master server for further distribution when a client is trying to find a match.

4.2.3 Game server access

As the game server is open for two players or certain players as in companies' friendly matches, the game server can't allow everyone to join the game room with every connection. One of the greatest issues without the access validation is that the room could have more clients than it was supposed to handle. That would lead to situations such as bugged out matches, network errors and disconnections. The solution for this issue is to have a game server access validation. This section explains the access pipeline to the game server's game room.

Handling the access happens on both, the client and the server-side. The client initiates the access request from the server, as seen in Figure 14, through a basic network message which is sent through the connection to the master server. MSF provides various MsfOpCodes for different network messages by default. To easily distinguish these server pipeline network messages from UNET's own ones, it is best to use MsfOpCodes instead of NetworkMessage codes. MSF does, however, support UNET's default MessageBase derived NetworkMessages (Barebones-Masterserver's Networking API 2017) if required.

```
1.  public void GetAccess(ROOM_PARAMETERS, CALLBACK_PARAMETER, CONNECTION_PARAMETER)
2.  {
3.      if (connection.IsConnected == false)
4.      {
5.          callback.Invoke(null, "Not connected");
6.          return;
7.      }
8.
9.      RoomAccessRequestPacket packet = new RoomAccessRequestPacket()
10.     {
11.         // ROOM_PARAMETERS are stored into RoomAccessRequestPacket as
12.         // defined by the needs of the room access.
13.         // Such parameters can be room id's and passwords.
14.     };
15.
16.     connection.SendMessage((short) MsfOpCodes.GetRoomAccess, packet, (status, response) =>
17.     {
18.         if (status != ResponseStatus.Success)
19.         {
20.             callback.Invoke(null, response);
21.             return;
22.         }
23.
24.         RoomAccessPacket access = response.Deserialize(new RoomAccessPacket());
25.         LastReceivedAccess = access;
26.
27.         callback.Invoke(access, null);
28.
29.         if (AccessReceived != null)
30.             AccessReceived.Invoke(access);
31.
32.         if (RoomConnector.Instance() != null)
33.             RoomConnector.Connect(access);
34.     });
35. }
```

Figure 14. MsfRoomsClient's GetAccess method.

Access to the game room is requested by the player clients. Initiating a match connects to the master server and fetches the best suited room candidate for the client through the match making functionalities. After finding a room to connect, and to be on the safe side, it was best to check that the master server connection was still intact to avoid network errors. Having validated the active connection, the RoomAccessRequestPacket DTO should be populated with the required parameters such as the room id, which without the master server cannot deduce the wanted game server the client wants to access. RoomAccessRequestPacket is sent through with the MsfOpCode GetRoomAccess network message to the master server, and an anonymous lambda callback is used to catch the response when the callback is invoked as illustrated in Figure 14. MsfOpCodes can be customized for developers' convenience, and the MsfOpCode sent can be a different one from GetRoomAccess, as long as the master server has a handler assigned for that

network message. Failure to have a handler will trigger a logged and handled error in the master server but leaves the message ignored.

Response from the room access handler can contain an error due to the DTO not correctly being transferred, invalid password if one is used, or no room was found with the given room id. The error state should be checked prior to further tries to establish the connection to the game server's game room. If trying to connect to the game server's game room with error message, the connection is more than likely going to throw a network error in the form of a timeout.

Successful response from the access request yields a RoomAccessPacket DTO, which is deserialized and sent down the pipeline of connecting to the game server through the access and the room connector callbacks as seen in Figure 14, which is the client-side requesting method for the room access.

Using the default RoomAccessRequestPacket and RoomAccessPacket DTOs are not mandatory, and a custom implementation can be done. In this phase of the game server implementation it was derived that it was not necessary, and the default packets were sufficient. Furthermore, addition of new properties to the DTOs could be done by plain variable declarations, reducing the need to create and test customized solutions. During the deployment of the master server pipeline, the room access DTOs worked as intended.

Having the client send the room access request to the game server's game room, a respond will be determined on the server-side by the master server. In Figure 15, this room access request handling is depicted further. The packet data needs to be deserialized into more readable form, and the room extraction try with the room id is initiated from the now constructed RoomAccessRequestPacket the deserialized packet gave.

```
protected virtual void HandleGetRoomAccess(IIncommingMessage message)
{
    RoomAccessRequestPacket data = message.Deserialize(new RoomAccessRequestPacket());

    RegisteredRoom room = null;
    Rooms.TryGetValue(data.RoomId, out room);

    if (room == null)
    {
        message.Respond("Room does not exist", ResponseStatus.Failed);
        return;
    }

    // If password is being used, validate it here and use ResponseStatus.Unauthorized
    // in case of a false password.

    // Respond room access request outcome to peer who sent it.
    room.GetAccess(message.Peer, data.Properties, (packet, error) =>
    {
        if (packet == null)
        {
            message.Respond(error, ResponseStatus.Unauthorized);
            return;
        }

        message.Respond(packet, ResponseStatus.Success);
    });
}
```

Figure 15. RoomsModule's HandleGetRoomAccess method.

In case the target room of the GetRoomAccess message exists, the peer info of the client is sent to the registered game room in the server pipeline, and an anonymous lambda method is used to catch the respond when it arrives through the callback method further down. The access possibility of the client for the wanted room, is deduced on the fly when the request arrives to the master server. With hundreds of thousands of players this can taxing for the master server. The process could be made run on multiple threads, but during the development there was not enough time for that. It is more than likely, that the server pipeline will be made to run concurrently on multiple threads with more asynchronous tasks in the future.

Final step to deduce the room access is to let the master server registered rooms distribute the accesses to themselves as seen in Figure 16. Mainly the client info parameter as an outside argument is used in this deduction. If the client is already in the room or the request is being progressed there is nothing to be done, the client must wait, and will be told to do so.

```
1.  public void GetAccess(IPeer peer, PROPERTIES, GetAccessCallback callback)
2.  {
3.      // Check requests in progress and contained players for the peer's id first.
4.      var currentAccess = GetUnconfirmedAccess(peer.Id);
5.      if (currentAccess != null)
6.      {
7.          // Refresh the currentAccess.Timeout with Options.AccessTimeoutPeriod.
8.          // Send the refreshed access back through the callback.
9.          callback.Invoke(currentAccess.Access, null);
10.         return;
11.     }
12.
13.     // Check for taken player slots.
14.     // Taken slots equal to existing and unconfirmed accesses, and in-progress requests.
15.     if (playerSlotsTaken >= Options.MaxPlayers)
16.     {
17.         callback.Invoke(null, "Room is already full");
18.         return;
19.     }
20.
21.     RoomAccessProvideCheckPacket packet = new RoomAccessProvideCheckPacket()
22.     {
23.         PeerId = peer.Id,
24.         RoomId =  RoomId
25.     };
26.
27.     // Add the request to pending requests list before sending the access check.
28.     Peer.SendMessage((short) MsfOpCodes.ProvideRoomAccessCheck, packet, (status, response) =>
29.     {
30.         // Remove the request from the pending requests list.
31.         if (status != ResponseStatus.Success)
32.         {
33.             callback.Invoke(null, response.AsString("Unknown Error"));
34.             return;
35.         }
36.
37.         RoomAccessPacket accessData = response.Deserialize(new RoomAccessPacket());
38.         RoomAccessData access = new RoomAccessData()
39.         {
40.             Access = accessData,
41.             Peer = peer,
42.             Timeout = DateTime.Now.AddSeconds(Options.AccessTimeoutPeriod)
43.         };
44.
45.         // Save the access as unconfirmed access.
46.         _unconfirmedAccesses[access.Access.Token] = access;
47.         callback.Invoke(access.Access, null);
48.     });
49. }
```

Figure 16. RegisteredRoom's GetAccess method.

It is also possible that the client-side has not confirmed the game server access due to possible packet loss or unfortunate miscalculation on the client-side. In that situation, the already existing access can be sent back with a refreshed timeout period. As a reminder, the process in Figure 16 can be done differently due to different requirements, design patterns and specific needs for the functionality. If the peer data is otherwise valid for the game server access, the limitations of maximum players in the match needs to be considered. To avoid unnecessary waiting times for the clients, the room is already full if

there are enough clients in the game server, or the combined amount of access requests in progress and unconfirmed accesses is enough to fill the room. The room will still be available if the accesses are not accepted, and the game server doesn't have enough clients to start the game.

In Figure 17, the room access checker that will double check the room existence and constructs access providers in the form of a RoomAccessPacket DTOs. MSF provides a default access provider if there are no specific needs to modify the RoomAccessPackets. The access packet mainly contains the game server IP and a port in addition to wanted properties.

```
1.  private void HandleProvideRoomAccessCheck(IIncommingMessage message)
2.  {
3.      RoomAccessProvideCheckPacket data = message.Deserialize(new RoomAccessProvideCheckPacket());
4.      RoomController roomController = Msf.Server.Rooms.GetRoomController(data.RoomId);
5.
6.      if (roomController == null)
7.      {
8.          message.Respond("No room controller for RoomId: " + data.RoomId, ResponseStatus.NotHandled);
9.          return;
10.     }
11.
12.     var accessProvider = roomController._accessProvider ?? DefaultAccessProvider;
13.     bool isProviderDone = false;
14.
15.     UsernameAndPeerIdPacket requester = new UsernameAndPeerIdPacket()
16.     {
17.         PeerId = data.PeerId,
18.         Username = data.Username
19.     };
20.
21.     accessProvider.Invoke(requester, (access, error) =>
22.     {
23.         if (isProviderDone)
24.             return;
25.
26.         isProviderDone = true;
27.         if (access == null) // Access was not provided.
28.         {
29.             message.Respond(error ?? string.Empty, ResponseStatus.Failed);
30.             return;
31.         }
32.
33.         message.Respond(access, ResponseStatus.Success);
34.     });
35.
36.     // If the access provider is a customized one and it is probable that it will take time,
37.     // it is best to create a timeout calculation request here.
38. }
```

Figure 17. RoomController's HandleProvideRoomAccessCheck method.

After the RoomAccessPacket is provided either as a successful access or not, the access validation is done, and the access results can be responded back to the client. Depending on the response, the client will join the match through a successful access or initiate a new request to another game server.

The room access pipeline, albeit long, is easily customizable for different scenarios. The current implementation is not perfected yet as can be seen from the logs received from the live game, where various players do get the “Room is already full” message internally. The denied clients will simply try to find another match until enough tries have been initiated and a room creating request is called on the master server due to unavailable rooms or just purely enough access failures. Getting into the match takes the longest if all find match tries have been used and a new game server is created. In the future, the finding of matches could be taken care of by having a master server room for the clients to wait until a match is found or a room is created for them. For the sakes of a convenient match making and accelerated finding of game servers, the client does not have to get into a room and after that check if the opponent is worthy as deemed through the match making logic. The rooms and players are filtered through the match making algorithm, thus saving the time for needless room accesses and disconnections due to an invalid opponent.

4.3 Networked objects

For synchronization and interaction purposes, the active objects in the network require network components to be valid networked objects. The game state of the match handles multiple objects from infantry units to buildings and various explosive components that need to be synchronized through the HLAPI. To be able to synchronize object states, these objects are required to have a component that derives from a NetworkBehaviour class and have a provided NetworkIdentity as well. This section of the implementation handles objects the networked game match requires with the purpose of introducing and reasoning their existence.

4.3.1 Player prefab

Each active network system player on the server is represented through player objects that are client instances with the ability to initiate command calls to the server. The HLAPI and the server can this way validate that the commands are issued from the correct client and connection.

Player object can be a prefab that is registered to the NetworkManager to allow default spawning of the player object when the client connects to the game, or the player object can be created by other means (Unity's Player Object 2017). For the game servers purposes the default way was deemed efficient and convenient enough as there were no specific needs to implement customized pipeline for creating the player object. The player object is imperative for the associated client connection as otherwise the client is unable to issue commands that are executed on the client connection associated player object on the game server-side (Unity's Player Object 2017).

In the game, the player objects were designed to be the central hub for handling the commands and inputs the clients would issue, such as spawning of card representative objects. This process would be supported through the match specific Match Manager object.

4.3.2 Match Manager object

Match Manager is used as a central hub for actions related to the match, whether they were caching of player specific resource points, sending of analytic events for development and marketing purposes, caching various match related statistics for later usage and numerous other match related tasks.

Client-side validation and a few elements in the battle UI are leaning on the data cached in the Match Manager. It is also an important tool when synchronizing the state of the match running on the clients while also handling the execution of various client issued remote commands.

Match Managers are Singleton instances and unique to each match. After the server scene of the game is loaded, prefab hosting the Match Manager is spawned, providing fresh values for each match. The current game implementation doesn't allow using the same Match Manager due destroying the object after each match.

4.3.3 Deployable cards

During the writing process of this thesis, there was a total of 41 different deployable cards available for players. Out of those cards, only a few were using multiple instances of related character, command or building prefabs. Spawnable prefabs were required to be registered intro NetworkManager due to HLAPI requirements for spawnable objects. To easily differentiate card entities, each character, command and building was stored as an individual prefab. The word character is used interchangeably between infantry and vehicles units because of shared implementations.

Each card is drawn to the battlefield issues a call to spawn a squad which is the connecting link between many operations available to characters and buildings alike. However, the usage of squads has its roots to the NGPlatform's game template, and is slowly turning out to be a legacy component pending for removal. Prior to removal and refactoring of key components, the squads are the first step in spawning the card entity into the battlefield. Spawned squads' references are registered and cached as each corresponding clients' property and are used to trigger the spawning of related characters, buildings or command agents.

Command agents are objects spawned by commands and they can take the form of a missile or even a building. To reduce the load of spawning single, multiple objects containing prefab instance, have a variable defining the amount of command agents to be spawned.

4.4 Game server match logic

Implementations of the match logic are technical and require knowledge of programming to fully grasp. Various match related scripted components are revealed and their reasonings explained as snippets of code are abstracted and partially written in pseudo to convey general insight and understanding into the codebase of Medals of War.

Generalization of code extractions was reasoned for the purposes of this thesis, allowing extended amount of shareable knowledge to be written. Full implementations can't be shown due to security and NDA reasons.

4.4.1 Match Manager logic

MatchManager is one of the largest scripts related to the functionality and coherence of the game server state. Focusing mainly on the server-side, MatchManager is void of its main functionalities on the clients and is there to ensure that the clients abide to the game server state through synchronization calls and RPC calls.

Illustrated in Figure 18, the MonoBehaviour provided Start method was overridden as a coroutine to halt the execution for a duration of the battle objective briefing to allow keeping the starting of the match in one method.

```
1.  IEnumerator StartMatchCoroutine()
2.  {
3.      SpawnNoMansLandSections();
4.      InitPoints();
5.
6.      matchStarting = true;
7.
8.      // Show battle objectives on the clients only.
9.      RpcShowBattleObjective ();
10.     yield return new WaitForSeconds(1.5f);
11.
12.     battleLength = CONFIG_BATTLE_LENGTH_VARIABLE;
13.     endTime = Time.realtimeSinceStartup + battleLength;
14.     _startTime = Time.realtimeSinceStartup;
15.
16.     matchStarting = false;
17.     matchStarted = true;
18.
19.     // Used to gather analytics data.
20.     NetworkPlayerData team0Player = NetworkPlayerHub.GetPlayerByTeam (0);
21.     NetworkPlayerData team1Player = NetworkPlayerHub.GetPlayerByTeam (1);
22.
23.     RequiredMatchAnalyticsData = new MatchAnalyticsData ();
24.     // Gather MatchAnalyticsData here!
25.
26.     // Server doesn't require to send analytic events.
27.     // Have the clients send their analytic event data with their ids.
28.     RpcSendAnalyticEvents (RequiredMatchAnalyticsData);
29.     ServerSetTeamBattlePerks ();
30.
31.     RequestMatchStart();
32.
33.     // Start manpower gain coroutine here!
34. }
```

Figure 18. MatchManager StartMatch coroutine.

To avoid unnecessary animations being run on the server, only the clients will execute the animation through a RPC call. NML sections do not exist on the server scene initially and are spawned in the starting process. Spawned NML

sections are registered on the server and client-sides which after the contestants' resource points are initialized and cached. Yielding the execution of the coroutine allows the clients focus on the objectives and to avoid the battle to start prematurely when the game is being initialized. After continuing the coroutine execution, the length of the battle is initialized through back-end sought variable and the analytically important variables are cached and sent to clients to be structured in more readable format. Analytical data is of struct type to allow default serialization over the network. In the end of the coroutine, clients' officer battle perks are cached and the request for match data is sent to the back-end, which in turn constructs the data in more iterative format and sends it back to the game server to be cached until the match ends.

It is important that the matches start in quick order and there are no unnecessary wait times after getting into battle. To have valid data being sent to analytics, it is important that the game server builds the data packets as can be seen in the coroutine in Figure 18.

Generation of manpower resource is done each frame by a coroutine. Rate of generation is calculated to cumulate one manpower after each back-end sought interval variable as seen in Figure 19.

```
1.    [ServerCallback]
2.    private IEnumerator BaseManPowerGainRoutine()
3.    {
4.        while (resultSequence == false)
5.        {
6.            float manpowerBaseGainAmount = (GameConfig.BASE_SUPPLY_REGEN * Time.unscaledDeltaTime / ManPowerGainRateTime);
7.            AddManPower(team0, manpowerBaseGainAmount);
8.            AddManPower(team1, manpowerBaseGainAmount);
9.
10.           yield return null;
11.       }
12.   }
```

Figure 19. MatchManager BaseManPowerGainRoutine coroutine.

Due to significant relation to the prevalent match state, resource generation is controlled by the game server. Separating the manpower gain routine from the Update method allows to control the intervals between resource gains instead of having to follow separate variable measuring current interval period in the Update method. To safeguard the coroutine, it is declared private with a server callback attribute.

Due to being one floating-point value, synchronizing the manpower value each frame is trivial to the bandwidth and processing tasks. Prior to being increased every frame, the manpower points were increased by one-third between routine intervals and this caused unnecessary complexity to the client-side prediction of available manpower points, leading to mislead observations of the available manpower after certain timeframe. Gradually generating manpower allows the players to assume the time between their next action more easily.

Synchronization of changed manpower values are handled automatically by UNET's SyncVar attribute on the manpower variable when the value has changed. To have the client instances of the game properly update the battle UI on the proper client, OnTeamManpowerChange method is an additional method hook attached to the SyncVar attribute to be invoked every time the synchronization is initialized on the clients as shown in Figure 20.

```
1.      [SyncVar(hook = "OnTeamManpowerChange")]
2.      public float TeamManpower = 0.0f;
3.
4.      private void OnTeamManpowerChange(float value)
5.      {
6.          if (TeamManpower == value)
7.              return;
8.
9.          TeamManpower = value;
10.         if (ManpowerChanged != null)
11.             ManpowerChanged (team, value);
12.
13.         if (PlayerInstance != null && PlayerInstance.Team == team && BattleHUD.Instance != null)
14.             BattleHUD.Instance.RefreshCurrentManpower (TeamManpower);
15.     }
```

Figure 20. MatchManager OnTeamManpowerChange method.

To ensure no needless UI refreshes are issued, a basic equal comparison is issued before continuing the execution of the OnTeamManpowerChange method. ManpowerChanged is a System.Action delegate to allow methods attached to the delegate to be called as needed. In the end, the battle UI is told to refresh the manpower bar at the bottom of the screen if the player instance's team is corresponding to the manpower related team.

Implementation-wise, command points are gained through similar methods as manpower. Instead of being incremented each frame, however, a command point is gained through card spawning. MatchManager caches these points and modifies the values based on client inputs. Same methods are used when

decreasing the points as well. The possible resource values are clamped between zero and the maximum resource value defined through the database.

Controlling every aspect of the game state, the match results must be equal among the clients and the server. There are three result states for the clients: victory, defeat and draw. For analytical purposes, match result variables were required to be relative to the match result. Match result is therefore determined by the game server, but the reward calculations are sent to be done on the back-end side as illustrated by Figure 21.

```
1.  [Server]
2.  internal void RequestMatchResults(System.Action<string> onResponseCallback)
3.  {
4.      if (_matchResultsRequested)
5.          return;
6.
7.      _matchResultsRequested = true;
8.
9.      // Calculate the winner here.
10.     // The winner is the destroyer of enemy HQ, or the winner
11.     // is the holder of the whole NML if the time runs out.
12.     // Otherwise no winners and the game is a draw.
13.
14.     // Create REQUEST_PARAMETERS here.
15.     // Update match related analytics here.
16.     // Create MATCH_RESULT_DATA_PARAMETERS here.
17.
18.     NetworkRequestManager.Instance.Request(REQUEST_PARAMETERS, (results) =>
19.     {
20.         JSONObject resultsObj = new JSONObject(results);
21.         // Finalize MATCH_RESULT_DATA_PARAMETERS here.
22.
23.         RpcMatchEndResults(MATCH_RESULT_DATA_PARAMETERS);
24.         if (onResponseCallback != null)
25.             onResponseCallback.Invoke(results);
26.     });
27. }
```

Figure 21. MatchManger RequestMatchResults method.

As the reward formula is not constant, having it reside over in the game server would require a new build distribution after every change. Having the formula on the back-end side is not without negative aspects however. If the back-end is under heavy load, the access to results panel can be stalled. It is required that only the correct match result is shown when the match ends. Failure to do so can result in drastically decreased UX. Fortunately, the back-end implementation can be improved and optimized without distributing a new client build.

In the previous host-to-client model, the host could discard opponent commands and have unlimited resources by tampering with the game. Due to this reason, it is mandatory to grant the game server authority over the clients and to have authoritative server hosting the match.

Having authority over the match state, various state affecting properties such as NML ownership states are internally accessible from the MatchManager. To sustain continue match state integrity, HLAPI allows server only objects to exist in the scene. These objects, such as spawn points and area colliders are in the direct control of the MatchManager. On the client-side, synchronized MatchManager distributed variables such as NML ownership states are available for clients to provide feedback for the players.

4.4.2 Player object Squad

HLAPI registered client connection can send and receive byte data and network messages over the network, but to be able to issue remote action commands, a player object is required by UNET as mentioned in the chapter Remote actions. Client remote command calls are always handled through the player object on the server-side and outputs on the client-side.

Deploying of cards through spawning was designed to be a straightforward pipeline. Initially the client-side constructs a DTO of the wanted card and validates spawning availability locally. Player objects are used to call a wrapper method for spawning different cards. These wrapper methods are shown in Figure 22.

```
1.      [Command]
2.      public void CmdSpawnUnitCardAt(UNIT_CARD_DATA, PATH_DATA, NetworkInstanceId ownerId)
3.      {
4.          NetworkSpawnTunnel.SpawnUnitCard (UNIT_CARD_DATA, PATH_DATA, ownerId);
5.      }
6.
7.      [Command]
8.      public void CmdSpawnCommandCardAt(COMMAND_CARD_DATA, PATH_DATA, NetworkInstanceId ownerId)
9.      {
10.         NetworkSpawnTunnel.SpawnCommandCard (COMMAND_CARD_DATA, PATH_DATA, ownerId);
11.     }
```

Figure 22. Squad SpawnCard methods.

The wrapper method allows both the game server and the client to call the same method where rest of the spawning pipeline is determined by basic

comparisons of peer privileges through the HLAPI variables isClient and isServer Booleans. A good example of what the game server doesn't require is the analytics events, which could be left out from the game server-side implementation. As seen in Figure 22, the game server is informed of the intent to spawn a card through command attributed spawn methods for units and commands.

When issuing an area of effect based command, the command agents were attached with a Vector3 argument accepting callback method activated on a collision event. Entities within the radius of the effect would be affected as defined by the command. As seen in Figure 23 the agent activation method is guarded with Server attribute to ensure only the server can issue the callback method.

```
1.      [Server]
2.      internal void OnActivateAgent(Vector3 pos)
3.      {
4.          if (MatchManager.instance.resultSequence == true || _activatedAgent == null)
5.              return;
6.
7.          float radius = _activatedAgent.AgentRadius;
8.          float damage = _activatedAgent.AgentDamage;
9.          List<IHealth> agentTargets = new List<IHealth>();
10.
11.         // Initialize a list containing all the NetworkBehaviours inside the activation radius.
12.         for (int healthIndex = 0; healthIndex  < HealthInstances.Count; ++healthIndex)
13.         {
14.             // Fetch Local reference for NetworkBehaviour health instance.
15.             IHealth iHealth = HealthInstances[healthIndex];
16.             ITeam iTeam = iHealth as ITeam;
17.             NetworkBehaviour behaviour = iHealth as NetworkBehaviour;
18.
19.             // Ignore remove pending null references.
20.             if (iHealth == null || iTeam == null || behaviour == null)
21.                 continue;
22.             // Ignore dead and non-hostile targets.
23.             if (iHealth.isAlive == false || iTeam.Team == Team)
24.                 continue;
25.
26.             // Check that the behaviour is inisde the agent radius.
27.             float distance = ToolKit.Distance2D(behaviour.cachedTransform.position, pos);
28.             if (distance < radius)
29.             {
30.                 agentTargets.Add(iHealth);
31.             }
32.         }
33.
34.         // Iterate over agentTargets list and apply the agent damage to them.
35.         // Register possible session stats related to the _activatedAgent's parent command.
36.     }
```

Figure 23. Squad's OnAgentActivation method.

To illustrate damaging area of effect methods, each health instance is cached in a list that is iterated to find valid damageable targets for the agent's damage and activation radius. Dependent on the command, the agent activation callbacks are modified to match the command requirements. Commands, such as Napalm Strike and Gas Strike, that share similarities have been abstracted to slightly differentiated activation methods.

The player object was deemed as a sufficient handler for client specific server-side outputs when processing client's input as this is well according to how Unity's own documentation suggests doing. HLAPI makes it mandatory to have exclusively defined player object in networked games or the client is unable to use commands as explained in chapter Remote actions.

4.4.3 Characters and buildings

Formerly mentioned Deployable cards includes characters and buildings that are AI controlled objects fighting for the clients with behavioral tasks in prioritized order: attack enemies in range, capture the NML section in the same side of the map and finally destroy the enemy HQ. This flow of behavioral tasks, especially with characters, can be affected through command cards Dig In and Rally during the writing of this thesis.

Characters and buildings consist of multiple scripts working together. Each weapon has a script that takes care of instantiating muzzle flash and tracers, while each object is using a special shooting AI to validate available targets and making the targets take damage. In addition, the namesake scripts of the character and building objects were created to be filled with wrapper and convenience methods to help simulate the match. Each character and building entity has its own behaviour tree consisting of behavioural tasks to differentiate their actions. The game server runs all the active behaviour trees, utilizing all the functionalities the characters and buildings offer in behavioural tasks. Due to being an enormous topic themselves, the AI and behaviour tree are only mentioned as one of the ruling factors when determining entities actions in the game world.

Being active and mutable objects, especially character entities require focused consideration to ensure smooth synchronization and interaction over networked instances of the match. Unity provides basic tools to help with this, though the community gives it the features of a non-trivial matter, as it seems to be one of the most asked questions related to networked objects.

Especially the synchronization of character movement is imperative for the UX and proper inspection of the match. Warping and stuttering characters are rather grievous to watch. Unity's HLAPI provides a NetworkTransform component as a solution. After basic prototyping, the component did not yield sufficient results. At its best, decreasing the send interval of UpdateVar packages and synchronizing the 3D Rigidbody Unity component did offer a measure of smooth synchronization. NetworkTransform does provide intermediate interpolation by sending predicted velocities through only position difference and ratio between the interpolation step and network send interval of UpdateVar packages (Unity's NetworkTransform 2017). This was not enough due to effortlessly perceivable warping and stuttering, and a custom implementation was derived as seen in Figure 24.

```
1.  [SyncVar(hook = "OnPosition")][HideInInspector]
2.  public Vector3 RealPosition = Vector3.zero;
3.  [SyncVar][HideInInspector]
4.  public Quaternion RealRotation = Quaternion.identity;
5.  [SyncVar][HideInInspector]
6.  public Vector3 RealVelocity = Vector3.zero;
7.
8.  private void TransformSync()
9.  {
10.     float positionDiff = ToolKit.Distance2D(cachedTransform.position, RealPosition);
11.
12.     Vector3 predictedMovement = RealVelocity * Time.unscaledDeltaTime;
13.     predictedTargetPosition += predictedMovement;
14.
15.     float positionFactor = Mathf.Clamp(positionDiff, 0.0f, 1.0f) + 1.0f;
16.     float lerpStep = (MovementSpeed * positionFactor * Time.unscaledDeltaTime);
17.     cachedTransform.position = Vector3.Lerp(cachedTransform.position, predictedTargetPosition, lerpStep);
18.
19.     float angleDiff = Quaternion.Angle(cachedTransform.rotation, RealRotation);
20.     float angleFactor = Mathf.Clamp(angleDiff, 1.0f, 180.0f);
21.     float slerpStep = (RotationSpeed * angleFactor * Time.unscaledDeltaTime);
22.     cachedTransform.rotation = Quaternion.Slerp(cachedTransform.rotation, RealRotation, slerpStep);
23.
24.     velocity = RealVelocity;
25. }
```

Figure 24. Characters TransformSync method.

To alleviate NetworkTransform's stuttering and warping, the movement synchronization was implemented through server-side caching of real positions,

rotations and velocities. The whole calculation can be seen in Figure 24. These server-side real values are enforced on the clients by the authoritative server model. To stem possible latency issues, client instances utilize interpolation between the real values and current values by calculating position and rotation differences and uses them to calculate the interpolation step every frame and not only when the UpdateVar packages have reached the client. This allows better predicted future position to alleviate the interpolation steepness in the next UpdateVar receival.

Arguably one of the most important interactions between characters and buildings is damage dealing. To avoid client tampering, the TakeDamage method responsible for distilling and dealing the damage amount, is guarded by server only attribute, disallowing client access. TakeDamage as seen in Figure 25, is a character side implementation. Due to significant similarities, it provides the idea behind the implementation.

```
1.  [Server]
2.  public void TakeDamage(float amount, IHealth damager = null)
3.  {
4.      if (MatchManager.instance == null || MatchManager.instance.resultSequence == true || isAlive == false)
5.          return;
6.
7.      if (IsInCover && _characterLevelStat != false)
8.      {
9.          amount *= _characterLevelStat.GetFloat(COVER_DAMAGE_MULTIPLIER);
10.     }
11.
12.     if (damager != null))
13.     {
14.         if (IsRallying == false && Squad.Type != ENGINEER)
15.         {
16.             NetworkBehaviour damagerBehaviour = damager as NetworkBehaviour;
17.             if (CanChaseTarget(damagerBehaviour))
18.             {
19.                 RpcSetChaseTarget(damagerBehaviour.gameObject);
20.             }
21.             Damager = damager;
22.         }
23.         if (PerkConditionDamage == 0.0f)
24.         {
25.             int opponentTeam = (team == 0) ? 1 : 0;
26.             Character damagerCharacter = damager as Character;
27.             if (damagerCharacter != null)
28.             {
29.                 // Calculate officer perk related damage.
30.             }
31.         }
32.     }
33.
34.     // Client feedback.
35.     RpcPlayHitSound();
36.     OnHealthChanged((CurrentHealth - amount));
37. }
```

Figure 25. Character's TakeDamage method.

Execution of the TakeDamage method is halted if the match is over, avoiding

unnecessary computations. Damage dealer is used to prompt the damaged party take counter actions and chase the target down if possible. Distilled damage is subtracted from the current health and sent down to On-HealthChanged method. State of the damage dealer doesn't affect the damage if damaging projectile has left the barrel, and is thus independent of the source after damaging action is triggered. Depending on the number of active entities of the match, the interval for TakeDamage calls can be abruptly short. Therefore, to avoid performance hits during networking, the damaging pipeline is needed to be light. Dealing with cached entity statistical variables and multipliers makes the pipeline fast and efficient as well.

More in related to the buildings, the buildings hitpoints decrease over time through decaying, which is a form of taking damage and falls into the same category of server only methods. Implementation of the decaying is depicted in Figure 26.

```
1.  [Server]
2.  private void UpdateDecay()
3.  {
4.      TimeFromLastDamage += Time.unscaledDeltaTime;
5.
6.      if (IsBuilt)
7.      {
8.          _intervalElapsed += Time.fixedDeltaTime;
9.
10.         if (_intervalElapsed > INTERVAL_DURATION && DecayDamage > 0.0f)
11.         {
12.             TakeDamage(DecayDamage);
13.             _intervalElapsed = 0.0f;
14.         }
15.     }
16. }
```

Figure 26. Building's UpdateDecay method.

Decaying logic is based on the concept of damage over time effects. Each decay interval, the building takes damage which is calculated to take down the building in certain number of seconds in a situation where the building doesn't receive external damage. Retaining the decaying logic on the server-side, the buildings won't be out of sync by decaying feature.

HQs of each client are already on the server scene loaded by clients. Being networked objects, in HLAPI this means that the objects are automatically spawned and assigned valid NetworkInstanceId when the sever scene is

loaded. Due to the HQs level is linked to the owner player, the client is required to be connected to the game properly by having his or her player object spawned into the scene. Without proper safe guards, the MatchManager can initiate the game start sequence without client's level data, causing misconceived level variables attached to the HQ.

To contend against mismatched level variables, the method responsible from assigning these variables required halting of execution in case of no valid data. The method is illustrated in Figure 27.

```
1.  IEnumerator Start()
2.  {
3.      while (Level <= 0)
4.      {
5.          while (MatchManager.instance == null)
6.              yield return null;
7.
8.          for (int squadIndex = 0; squadIndex < Squad.AllInstances.Count; ++squadIndex)
9.          {
10.             Squad squad = Squad.AllInstances[squadIndex];
11.             if (squad.isAuthorSquad == false)
12.                 continue;
13.
14.             // Find corresponding team player object for the HQ.
15.             if (Team == squad.Team)
16.             {
17.                 Level = squad.PlayerLevel;
18.                 break;
19.             }
20.         }
21.
22.         yield return null;
23.     }
24.
25.     SetLevelStats(Level);
26. }
```

Figure 27. HQ Start coroutine.

During the testing, the problem seemed to be from a couple of hundred milliseconds to a few seconds. In relation to the waiting delay of client's proper connection it was trivial, but regarding mismatched levels it was quite significant. It is beneficial to use coroutines to handle the tasks that require arbitrary data that might not be loaded when calling the execution of the method. The Start coroutine of HQs, as seen in Figure 27, merely waits for matching team indexed player object to appear, allowing it to assimilate the player's level when fetching the correct level data for the HQ.

It was tempting to use SyncVar variables on various occasions during the development. However, HLAPI has a hard cap of 32 SyncVar attributes on one class whether all of them being on one class or combination of classes through polymorphism. It must be noted that having continuous synchronization and SyncVar hook calls from multiple representatives of the class does increase the network traffic considerably. To alleviate these issues, various SyncVar variables were discarded of their attributes and were refactored to work without them during the development. It is practical to synchronize variables that are related to the perceived game of the client and variables such as internal timings, if not communicated to the player, are best left out from synchronized variables.

5 CONCLUSIONS

This thesis was meant to describe an implementation of Unity and MSF based game server which is currently in an active use going through modifications to the structure and logic. The processes and implementation sections in this thesis introduced the game server pipeline and what needs to be considered when developing a dedicated game server. The finished product can be considered as a success.

As stated in the beginning of this thesis (Functional Requirements chapter), there were certain requirements for the game server to abide by. During the final testing process, all the requirements were met. The clients were not able to perform any invalid actions that would affect the state of the match. Implemented functionalities did not bloat the bandwidth, and the match specific data was properly bound to the game server and communicated with the back-end.

On its own, this thesis can act as a documentation about implementing MSF based game server logic and even as an exemplary for UNET and MSF implementations. The game server logic contains starting the game server, connection to the closest or parameter-bound master server, registering to the master server and giving access to the game server’s game room. More specific to the game Medals of War and its matches, the match logic is described and dissected to document the code structures and implementations of the match and how they serve the set requirements.

As a project, the writing of this thesis was an extremely interesting task. With no prior knowledge of UNET and MSF in general, working with both of them was enlightening, rewarding and taught a lot. MSF provided documentation gave quite a lot headwork on the internal workings of the game server, but to get everything to work properly was challenging. Tremendous support from the Nitro Games senior personnel was imperative in getting the game server to work the way it was required, and the insights given from them were learning experiences on their own.

To have a good structure for the future was important as having the game being live and playable in the public makes it more difficult to implement new features and functionalities. The learnings made during the development have given a lot of structure for future modifications, which are mainly optimizations and increased scalability in functionalities. Future game server implementations for other projects are most likely easier when done in conjunction with this thesis due to the amount of knowledge and the descriptions of functionalities provided.

REFERENCES

Barebones-Masterserver 2017. Networking API. WWW document. Available at: https://github.com/alvyxaz/barebones-masterserver/wiki/Networking-API [Accessed 10 October 2017].

Barebones-Masterserver 2017. Peer Extensions and Properties. WWW document. Available at: https://github.com/alvyxaz/barebones-masterserver/wiki/Peer-Extensions-and-Properties [Accessed 10 October 2017].

Ignatchenko, S. 2017. Development and Deployment of Multiplayer Online Games, Volume 1. Leanpub.

Ignatchenko, S. 2017. Development and Deployment of Multiplayer Online Games, Volume 3. 2nd beta. Leanpub.

Glazer, J., Madhav, S. 2015. Multiplayer Game Programming: Architecting Networked Games (Game Design). 1st ed. Boston: Addison-Wesley Professional.

Nystrom, R. 2014. Game Programming Patterns. 1st ed. Genever Benning.

Medals of War. 2017. Medals of War. WWW document. Available at: https://www.medalsofwargame.com/ [Accessed 1 August 2017].

Pandya, K. 2013. International Journal of Advance Research in Computer Science and Management Studies. [Research Paper] Network Structure or Topology, Volume 1, Issue 2. Available at: www.ijarcsms.com/docs/paper/volume1/issue2/v1i2-0006.pdf [Accessed 2 August 2017].

Poškevičius, A. 2017. Introduction To The Framework. GitHub barebones-masterserver repository's ReadMe. WWW document. Available at: https://github.com/alvyxaz/barebones-masterserver/wiki/Introduction-To-The-Framework [Accessed 10 August 2017].

Schollmeier, R. 2002. A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. Proceedings of the First International Conference on Peer-to-Peer Computing. WWW document. Available at: http://www.it.lut.fi/wiki/lib/exe/fetch.php/courses/ct30a6900/p2p_definitions.pdf [Accessed 3 August 2017].

Tanenbaum, A. S., Wetherall, D. J. 2010. Computer Networks. 5th ed. Prentice Hell.

Unity Documentation 2017. Advanced operations: Using the LLAPI. WWW document. Available at: https://docs.unity3d.com/Manual/UnityWebRequest-LLAPI.html [Accessed 17 September 2017].

Unity Documentation 2017. ClientRpcAttribute. WWW document. Available at: https://docs.unity3d.com/ScriptReference/Networking.ClientRpcAttribute.html [Accessed 15 September 2017].

Unity Documentation 2017. CommandAttribute. WWW document. Available at: https://docs.unity3d.com/ScriptReference/Networking.CommandAttribute.html [Accessed 15 September 2017].

Unity Documentation 2017. Creating DownloadHandlers. WWW document. Available at: https://docs.unity3d.com/Manual/UnityWebRequest-CreatingDownloadHandlers.html [Accessed 27 October 2017].

Unity Documentation 2017. Network Lobby Manager. WWW document. Available at: https://docs.unity3d.com/Manual/class-NetworkLobbyManager.html [Accessed 19 September 2017].

Unity Documentation 2017. Object Spawning. WWW document. Available at: https://docs.unity3d.com/Manual/UNetSpawning.html [Accessed 15 September 2017].

Unity Documentation 2017. Player Objects. WWW document. Available at: https://docs.unity3d.com/Manual/UNetPlayers.html [Accessed 19 September 2017].

Unity Documentation 2017. State Synchronization. WWW document. Available at: https://docs.unity3d.com/Manual/UNetStateSync.html [Accessed 15 September 2017].

Unity Documentation 2017. The High Level API. WWW document. Available at: https://docs.unity3d.com/Manual/UNetUsingHLAPI.html [Accessed 15 September 2017].

Unity Technologies 2017. Unity Networking NetworkTransform source code (Commit bb1e71). [Source code] Available at: https://bitbucket.org/Unity-Technologies/networking/src/bb1e7e19569365b8b95ef5dcff63b213e77525a8/Runtime/NetworkTransform.cs?at=2017.1&fileviewer=file-view-default [Accessed 19 September 2017].

Unity Documentation 2017. UnityWebRequest.SendWebRequest. WWW document. Available at: https://docs.unity3d.com/ScriptReference/Networking.UnityWebRequest.SendWebRequest.html [Accessed 27 October 2017].

Valve Developer Community 2001. Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization. WWW document. Available at: https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization [Accessed 2 August 2017].

Weilbacher, M. 2012. Dedicated Servers in Gears of War 3: Scaling to Millions of Players. [PowerPoint presentation] Scaling to Millions of Players. Microsoft Studios, GDC 2012. 4 March. Available at: http://www.gdcvault.com/play/1015559/Dedicated-Servers-In-Gears-of [Accessed 2 August 2017].

Xicota, D. 2013. Lag compensation tecniques for multiplayer games in realtime. Article 19.1.2016. Gamedonia. WWW document. Available at: http://www.gamedonia.com/blog/lag-compensation-techniques-for-multiplayer-games-in-realtime [Accessed 3 August 2017].

LIST OF FIGURES

Figure 1. CMD package.

Figure 2. The server structure and pipeline of MSF. The Big Picture (Poškevičius 2017).

Figure 3. GameServer Awake method.

Figure 4. GameServer WaitConfigs coroutine.

Figure 5. ConnectionToMaster's InitConnection coroutine.

Figure 6. GameServer's OnConnected method.

Figure 7. ConnectionToMaster's StartSpawned method.

Figure 8. UnetGameRoom's OnStartServer method.

Figure 9. UnetGameRoom's BeforeRegisteringRoom method.

Figure 10. UnetGameRoom's RegisterRoom method.

Figure 11. MsfRoomsServer's RegisterRoom method.

Figure 12. RoomsModule's HandleRegisterRoom method.

Figure 13. RoomModule's RegisterRoom method.

Figure 14. MsfRoomsClient's GetAccess method.

Figure 15. RoomsModule's HandleGetRoomAccess method.

Figure 16. RegisteredRoom's GetAccess method.

Figure 17. RoomController's HandleProvideRoomAccessCheck method.

Figure 18. MatchManager StartMatch coroutine.

Figure 19. MatchManager BaseManPowerGainRoutine coroutine.

Figure 20. MatchManager OnTeamManpowerChange method.

Figure 21. MatchManger RequestMatchResults method.

Figure 22. Squad SpawnCard methods.

Figure 23. Squad’s OnAgentActivation method.

Figure 24. Characters TransformSync method.

Figure 25. Character's TakeDamage method.

Figure 26. Building's UpdateDecay method.

Figure 27. HQ Start coroutine.